



PROGRAMMING Expert

This month **Mike James** gets extremely animated as he explains how to transform a series of bitmap images into an AVI video file using the Video for Windows API and the necessary programming



AJAX CLEANS UP

Asynchronous JavaScript with XML (AJAX) is an exciting new web programming technology behind a number of well-known websites, including Google Maps and Yahoo!'s Flickr.com photo-sharing site.

AJAX combines the best of two types of programming used in web pages: client-side and server-side scripts. Client-side script runs in the viewer's browser, which means it's fast but has difficulty getting live information from the server. Server-side scripts run on the web server, meaning they can access all the data on the server and send nothing but standard HTML web pages to the client. AJAX uses client-side JavaScript to make a web page work fast, but it can also make a direct XML call to the server when more data is needed. Microsoft has an implementation, called ATLAS, in the works. For details, visit www.ajaxmatters.com.

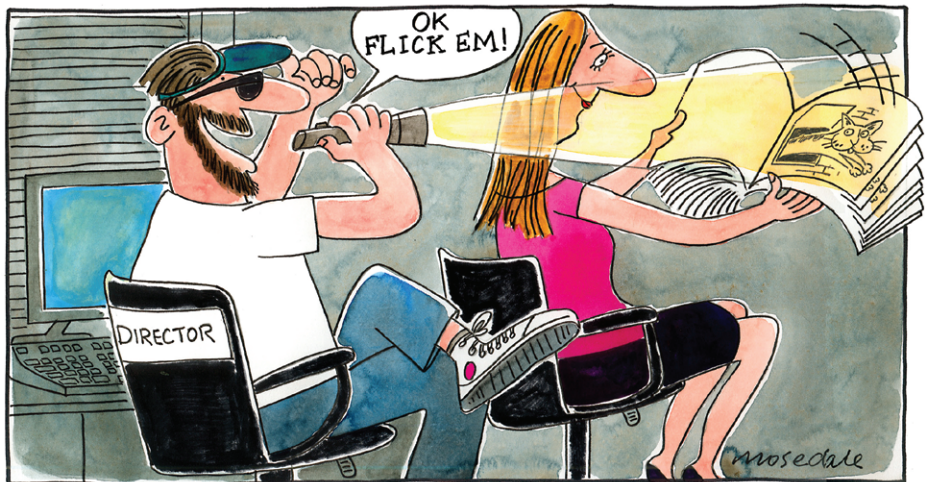
RUBY GETS UPDATE

The open-source programming language Ruby has been updated for web developers. Ruby on Rails, or RoR, includes a framework that allows Ruby to be used to create smart web pages. It claims to have been designed to make life easy for programmers rather than the compiler.

Ruby has gained a lot of support because it's easy to use, object oriented and free. It is similar to Java and C#. It's available for Linux and OS X, and there is also an easy-to-install version for Windows. If you aren't satisfied with ASP .NET, Python, PHP or any of the other ways of programming a web page, this is worth a look. See www.ruby-lang.org for details.

XML PATENT PROBLEM

American company Scientigo claimed to have two patents that cover XML's basic idea of storing data in a self-defining manner. If successful, it could demand a licence fee for software incorporating XML. This would increase the cost of software such as the next version of Microsoft Office, which plans to use XML as an open format for documents, and class libraries such as .NET and Java that provide XML facilities. Visit www.scientigo.com.



A basic technique used in animation is the conversion of a series of still images into a video file. If you've ever wanted to create your own animations for Windows, this project is for you. Here we'll turn a series of bitmap images into an AVI video file. Along the way, we'll learn about file formats and working with Windows APIs.

Microsoft provides the Windows Media Format SDK for manipulating digital media files, but at the moment this is hard to work with in any language other than C++. Eventually Microsoft will no doubt create a managed interface for the SDK and it will be easy to use it from all the .NET languages. Meanwhile there is a good alternative in the form of Video for Windows (VfW). This is the original video system introduced for Windows 98. Although it has been largely overtaken by Windows Media, it is still supported. If you want to create an AVI video file, VfW is adequate and has the advantage of being available on older computers.

There are some problems with using VfW, the first being its terrible documentation. You can read about VfW at http://msdn.microsoft.com/library/en-us/multimed/html/_win32_video_for_windows.asp or search for video for Windows on Microsoft's website (www.microsoft.com). Also, using VfW involves making API function calls, and this requires you to transfer data from whichever high-level language you are using to the API's primitive data types. This subject will be familiar to any Visual Basic 6 programmer, but it's new territory for those using .NET languages such as C#. If you want to see an example of how to use VfW from VB 6, look at www.shrinkwrapvb.com.

We will use C# for our project. Beta 2 of the C# Express programming environment can be downloaded free from Microsoft's website. Using VB .NET is very similar as the process relies heavily on using the .NET framework, which is common to

all .NET languages. As part of our project, we'll take in the following programming topics: creating a new class to read in a bitmap file; using structs in API calls; reading structs from disk; calling API functions from C#; using managed memory in API calls and using the Video for Windows API.

A BITMAP CLASS

The first problem we face is reading in information from bitmap files. You may think this would be easy as there is a ready-made Bitmap class as part of the GDI+. But this hides all the details of the bitmap needed to use the VfW API. The ideal solution is to extend the Bitmap class using inheritance, but for some reason Microsoft has decided not to let you do this, marking the class as sealed. This goes against one of the big ideas of object-oriented programming: reusing code. Happily, creating a custom RawBitmap class from scratch is easy.

Start a new Windows application and add a Class to the project called RawBitmap. A bitmap file always starts the same – a BITMAPFILEHEADER followed by a BITMAPINFOHEADER. Next comes an RGBQUAD array, which specifies the image's palette if one is needed, and then comes the array of colour data that makes up the image. To read in a bitmap, we just define C# Structs for each bit of the file and read them in. You'll find the definitions for the structures in C++ in the documentation on the Microsoft website. Translating them to C# is quite easy, but there are a couple of subtle points. For example, the BITMAPFILEHEADER struct is:

```
[StructLayout(LayoutKind.Sequential,
    Pack = 2)]#
public struct BITMAPFILEHEADER#
{#
    public Int16 bfType;#
    public Int32 bfSize;#
```



```
public Int16 bfReserved1;
public Int16 bfReserved2;
public Int32 bfOffBits;
};
```

Compare this to the C++ definition; you'll see the C# variable types Int16 and Int32 are substituted for the C++ WORD and DWORD types. The original variable names, although clumsy, have been retained, as it makes interpreting the documentation easier. If you don't know how to declare a struct in C#, you may not understand the purpose of the line:

```
[StructLayout(LayoutKind.Sequential,
    Pack = 1)]
```

Normally the .NET compiler will arrange the components of the struct in memory to make it faster to access. The StructLayout attribute allows you to control its layout to be compatible with existing data structures. In this case the Sequential attribute tells it to place the data types in memory one after another as they appear in the definition. Pack=1 tells it to make sure the struct is packed so it starts on a 1 byte address boundary. This ensures no extra padding bytes are added. As the default is Pack=16 and the struct needs a minimum of 14 bytes to store, the compiler adds two extra bytes to make it 16 bytes. This is not helpful when you are trying to work with structures stored on disk that do not have the packing bytes.

To make all this work, start a new Windows application and add a Class to the project called RawBitmap. To the standard 'using' lines, add:

```
using System.Runtime.InteropServices;
```

Add the struct for BITMAPFILEHEADER. The BITMAPINFOHEADER is translated to C# as:

```
[StructLayout(LayoutKind.Sequential,
    Pack = 1)]
public struct BITMAPINFOHEADER
{
    public int biSize;
    public int biWidth;
    public int biHeight;
    public short biPlanes;
    public short biBitCount;
    public int biCompression;
    public int biSizeImage;
    public int biXPelsPerMeter;
    public int biYPelsPerMeter;
    public int biClrUsed;
    public int biClrImportant;
};
```

Next we need a method to read the raw bytes in the bitmap file into the structs. We must open a FileStream, read bytes into a buffer and transfer the bytes into the appropriate structs. You can create a function that will read a struct of any type without using advanced features of C# 2.0, such as generics. The following must be inserted in class RawBitmap. It can only return an object type because it doesn't know what the struct will be before it is called:

```
private object ReadStruct(FileStream
    fs, Type t)
{
    byte[] buffer = new
```

```
byte[Marshal.SizeOf(t)];
    fs.Read(buffer, 0,
        Marshal.SizeOf(t));
    GCHandle handle =
        GCHandle.Alloc(buffer,
            GCHandleType.Pinned);
    Object temp =
        Marshal.PtrToStructure(
            handle.AddrOfPinnedObject(), t);
    handle.Free();
    return temp;
}
```

This function creates a byte array of the same size as the struct using the Marshal object, which has lots of useful methods for converting from C++ to C# types. Next the bytes are read into the buffer. To transfer these to the struct, we must pin the byte array so the Garbage Collector (GC) doesn't move it. Once we have a handle to the pinned object, we can get its address and pass it to the PtrToStructure method. This reads the bytes from the byte array and uses them to create a new structure of type t, typed as an object. This is returned by the function.

To write a LoadFromFile method we need public storage for the structs and bitmap data:

```
class RawBitmap
{
    public BITMAPFILEHEADER bmFH =
        new BITMAPFILEHEADER();
    public BITMAPINFOHEADER bmIH =
        new BITMAPINFOHEADER();
    public byte[] bits;
```

The new method simply reads the structures in using the function given above:

```
public void LoadFromFile(string
    filename)
{
    FileStream fs = new FileStream(
        filename, FileMode.Open,
        FileAccess.Read);
    bmFH = (BITMAPFILEHEADER)
        ReadStruct(
            fs, typeof(BITMAPFILEHEADER));
    bmIH = (BITMAPINFOHEADER)
        ReadStruct(
            fs, typeof(BITMAPINFOHEADER));
```

To get the bitmap bits we use the information in the bitmap header to dimension the byte array and then read them:

```
fs.Position = bmFH.bfOffBits;
bits = new byte[bmIH.biSizeImage];
fs.Read(bits, 0, bits.GetLength(0));
fs.Close();
}
```

The bitmap data might not start straight after the two structs because of variations in the bitmap format. This is why we've used the information in the file header to position the file to the start of the data. To make it all work, we also need to add:

```
using System.IO;
```

MAKING A MOVIE

Now we have a bitmap class, we can move on to make use of it within a BmpToAVI class. Add a new

class called BmpToAVI to the project. The new class will use a number of API calls within the VFW API, but rather than translate them all it makes sense to create definitions only for those that are needed. When you use VFW, you must initialise the system and when you have finished you have to free it. These actions are best done in the class constructor and destructor, as these are called automatically when the class is created and destroyed:

```
class BmpToAVI
{
    [DllImport("avifil32.dll")]
    extern static void AVIFileInit();
    [DllImport("avifil32.dll")]
    extern static void AVIFileExit();

    public BmpToAVI(IntPtr hwnd)
    {
        m_hWnd = hwnd;
        AVIFileInit();
    }

    ~BmpToAVI()
    {
        AVIFileExit();
    }
}
```

The need for the hwnd parameter will become apparent shortly. For now, we simply store it in a private variable declared at the top of the class:

```
private IntPtr m_hWnd;
```

The API definitions are the first of several, and they define the DLL in which the functions reside. Things get more complicated when we have parameters to translate to C# data types. To make these DllImport commands work, we need to add:

```
using System.Runtime.InteropServices;
```

The Vfw AVI system works with multimedia data streams stored inside a single file. Streams can be video, audio or text. We have to open and create an AVI file; this can be done as part of adding the first bitmap to the video. Add the following to the class:

```
public void FirstFrame(
    string AVIfile,
    string BMPfile)
{
    int result = AVIFileOpen(ref pFile,
        AVIfile, OF_WRITE | OF_CREATE,
        0);
```

This returns a pointer to the AVI file that is used in other API calls. Like m_hWnd, this must be declared as a private global variable along with the other pointers to the streams that we are going to create – one standard stream and one compressed stream:

```
private IntPtr pFile = IntPtr.Zero;
private IntPtr pStream = IntPtr.Zero;
private IntPtr psComp = IntPtr.Zero;
```

Now we define the API call used above:

```
[DllImport("avifil32.dll")]
extern static int AVIFileOpen(
    ref IntPtr pfile, string File,
    int Mode, int clsidHandler);
```



REVIEW BOOK

ASP.NET 2.0:
A Developer's Notebook

RATING ★★★★★

PRICE £21

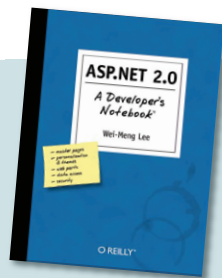
SUPPLIER www.amazon.co.uk

ISBN 0596008120

PAGES 256

ASP.NET 2.0 is Microsoft's server-side language for building sophisticated web pages. It's new, but many programmers already know versions 1.1 and 1.2. Author Wei Meng Lee's book is aimed at bringing experienced ASP users up to speed.

Each short section of the book deals with a single topic. The selection of topics is fairly arbitrary. Although all are potentially useful, there's no real logic to the presentation. There isn't much discussion of general principles, either. The cover explains that the book is "All lab – no lecture".



The examples are quite difficult to follow and are presented as listings, without being woven into the text. They use only VB.NET, but this shouldn't be a big problem. Topics covered include the Web Parts Framework, security, membership APIs and new controls such as

GridView and data source. There is also a discussion of the overall structure of a website, including Master pages and skins.

This is a worthwhile addition to your bookshelf if you already know the basics of ASP and just want something to dip into as and when the need arises, but it isn't comprehensive. Its biggest drawback is that it fails to impart wider principles to help you work things out for yourself.



A practical account of ASP.NET 2.0



No deep insights; illogical organisation

and the constants we used are defined as:

```
private const int OF_
WRITE=0x00000001;
private const int OF_
CREATE=0x00001000;
```

The values for constants and error codes can be found in the file VFW.h. This is contained in the platform SDK, which can be downloaded from Microsoft's website. The translation of the DLL function is relatively easy. The IntPtr type is useful for almost any 32-bit handle or pointer. The result returned should be zero if the function has worked.

Now we have the file open, we have to create a video stream to store in it. We must provide information about the format of the video. This isn't difficult, but the API seems to need this or similar information supplied to it more often than seems necessary. Much of the format information is concerned with the details of the bitmaps that will be used to create the stream, so now is a good time to use our RawBitmap class to load the first bitmap:

```
RawBitmap bm = new RawBitmap();
bm.LoadFromFile(BMPfile);
```

Next we need to fill in the details in an AVISTREAMINFO struct. This is easy to translate from the C++ definition:

```
StructLayout(LayoutKind.Sequential,
Pack = 1)]
public unsafe struct AVISTREAMINFO
{
    public Int32 fccType;
    public Int32 fccHandler;
    public Int32 dwFlags;
    public Int32 dwCaps;
    public Int16 wPriority;
    public Int16 wLanguage;
    public Int32 dwScale;
    public Int32 dwRate;
    public Int32 dwStart;
    public Int32 dwLength;
    public Int32 dwInitialFrames;
    public Int32
```

```
dwSuggestedBufferSize;
    public Int32 dwQuality;
    public Int32 dwSampleSize;
    public AVI_RECT rcFrame;
    public Int32 dwEditCount;
    public Int32 dwFormatChangeCount;
    public fixed char szName[64];
};
[StructLayout(LayoutKind.Sequential,
Pack = 1)]
public struct AVI_RECT
{
    public Int32 left;
    public Int32 top;
    public Int32 right;
    public Int32 bottom;
};
```

The only complication is the need to use a fixed-size character array, szName[64]. This is new in C# 2.0 and is considered unsafe, so you have to remember to allow unsafe code in the project properties. The AVISTREAMINFO struct contains an AVI_RECT struct, hence the second definition.

Here's how to load the details into the structure. The only difficulty is knowing which parameters are important and which are not:

```
AVISTREAMINFO Sinfo = new
AVISTREAMINFO();
Sinfo.fccType = mmioStringToFOURCC(
"vids", 0);
Sinfo.fccHandler = 0;
Sinfo.dwScale = 1;
Sinfo.dwRate = 10;
Sinfo.dwSuggestedBufferSize =
bm.bmIH.biSizeImage;
Sinfo.rcFrame.top = 0;
Sinfo.rcFrame.left = 0;
Sinfo.rcFrame.right =
bm.bmIH.biWidth;
Sinfo.rcFrame.bottom =
bm.bmIH.biHeight;
```

It is clear we need to specify the type of the stream, "vids" for video in this case, the size of the frame

and the frame rate, set to 10 frames per second here. We must also import the mmioStringToFOURCC function, which converts the string "vids" into the multimedia code for a video stream:

```
[DllImport("winmm.dll", EntryPoint =
"mmioStringToFOURCC")]
extern static int mmioStringToFOURCC(
string sz, int Flags)
```

In this case, we use an EntryPoint to specify the function in the DLL because it has a slightly different name to the C# function. The stream can be created using the format information:

```
result = AVIFileCreateStream(
pFile, ref pStream, ref Sinfo);
```

This returns a pointer to the video stream within the AVI file. The definition of the API call is:

```
[DllImport("avifil32.dll")]
extern static int AVIFile
CreateStream(
IntPtr pfile, ref IntPtr pavi,
ref AVISTREAMINFO lParam);
```

GETTING THE COMPRESSION

Now we have a stream we could start storing bitmaps in it, but few video players – and certainly not Windows Media Player – can play an uncompressed stream. We must create a compressed stream from the uncompressed stream. The simplest way is to let the user select compression settings from a dialog box provided by the API. This is why we needed the hwnd parameter in the constructor to allow the dialog box to have a reference to its parent window. First we need a structure to define the compression options:

```
[StructLayout(LayoutKind.Sequential,
Pack = 1)]
public struct AVICOMPRESSOPTIONS
{
    public Int32 fccType;
    public Int32 fccHandler;
    public Int32 dwKeyFrameEvery;
    public Int32 dwQuality;
    public Int32 dwBytesPerSecond;
    public Int32 dwFlags;
    public Int32 lpFormat;
    public Int32 cbFormat;
    public Int32 lpParms;
    public Int32 cbParms;
    public Int32 dwInterleaveEvery;
};
```

The API function that displays the dialog is defined as:

```
[DllImport("avifil32.dll")]
extern static int AVISaveOptions(
IntPtr hwnd, int uiFlags,
int noStreams, IntPtr ppavi,
ref IntPtr ppOptions);
```

The last two parameters are different from anything used so far and likely to cause trouble if you don't treat them correctly. Both ppavi and ppOptions are pointers to pointers. As a result, we have to pin the memory areas to which they refer to stop them being moved by the Garbage Collector. We also need some unmanaged memory for the API



function to use to store the user's selections. This can be provided using the `AllocHGlobal` method of the `Marshal` object:

```
IntPtr buf = Marshal.AllocHGlobal(
    Marshal.SizeOf(typeof(AVICOMPRESS
        OPTIONS)));
```

Next we need to get a handle, essentially a pointer, to this area of memory and pin it:

```
GCHandle handle1 = GCHandle.Alloc(
    pStream, GCHandleType.Pinned);
```

We can call the API function:

```
result = AVISaveOptions(m_hWnd,
    ICMF_CHOOSE_KEYFRAME |
    ICMF_CHOOSE_DATARATE, 1,
    handle1.AddrOfPinnedObject(),
    ref buf);
```

The constants used are:

```
private const int ICMF_CHOOSE_
    KEYFRAME = 0x00000001;
private const int ICMF_CHOOSE_
    DATARATE = 0x00000002;
```

Finally we transfer the selection made by the user to the structure and free the memory and the handle to the memory:

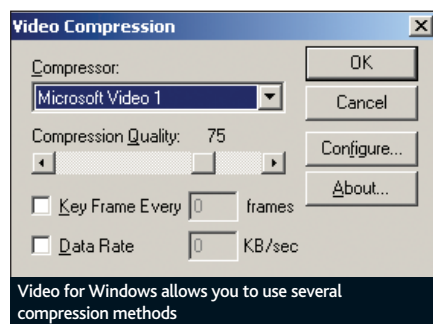
```
AVICOMPRESSOPTIONS opts =
    (AVICOMPRESSOPTIONS)
    Marshal.PtrToStructure(buf,
        typeof(AVICOMPRESSOPTIONS));
Marshal.FreeHGlobal(buf);
handle1.Free();
```

If the user presses the OK button, the result of the function is one.

Now we must create the compressed stream using the options selected by the user and then set its format. The two functions needed for this are:

```
[DllImport("avifl32.dll")]
extern static int
    AVIMakeCompressedStream(
        ref IntPtr ppsComp,
        IntPtr ppavi,
        ref AVICOMPRESSOPTIONS opts,
        int pClisidHandler);

[DllImport("avifl32.dll")]
extern static int
    AVIStreamSetFormat(
        IntPtr ppavi, int lpos, IntPtr
        lpFormat, int cbFormat);
```



To make the compressed stream we simply use:

```
result = AVIMakeCompressedStream(
    ref psComp, pStream, ref opts, 0);
handle = GCHandle.Alloc(bm.bmIH,
    GCHandleType.Pinned);
```

This returns a pointer, `pStream`, to the compressed stream. To set the format of the compressed stream we need to use the bitmap information header. This needs to be pinned before being passed to the API:

```
handle = GCHandle.Alloc(bm.bmIH,
    GCHandleType.Pinned);
result = AVIStreamSetFormat(psComp,
    0, handle.AddrOfPinnedObject(),
    bm.bmIH.biSize);
handle.Free();
```

The final API call writes the first bitmap's data to the compressed stream. This too has to be pinned before use:

```
handle = GCHandle.Alloc(bm.bmIH,
    GCHandleType.Pinned);
result = AVIStreamWrite(psComp,
    1, 1, handle.AddrOfPinnedObject(),
    bm2.bits.GetLength(0),
    AVIIF_KEYFRAME, 0, 0);
handle.Free();
```

We also need a global variable to keep track of the number of frames written and a constant definition:

```
private int FrameCount = 0;
private const int AVIIF_KEYFRAME =
    0x00000010;
```

After writing one frame, we set this to 1:

```
FrameCount = 1;
```

The definition of the API function is:

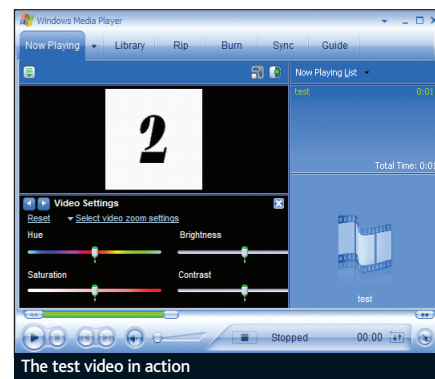
```
[DllImport("avifl32.dll")]
extern static int AVIStreamWrite(
    IntPtr ppavi, int lStart,
    int lSamples, IntPtr buffer,
    int cbBuffer, int dwFlags,
    int sampwritten, int bytes);
```

ADDING FRAMES AND CLEANING UP

The `FirstFrame` method is long because as well as storing the first frame it has to set everything up. The `NextFrame` method is simpler; we just have to read the next bitmap and write it to the stream:

```
public void NextFrame(string BMPFile)
{
    RawBitmap bm = new RawBitmap();
    bm.LoadFromFile(BMPFile);
    GCHandle handle = GCHandle.Alloc(
        bm.bits, GCHandleType.Pinned);
    int result = AVIStreamWrite(psComp,
        ++FrameCount, 1,
        handle.AddrOfPinnedObject(),
        bm.bits.GetLength(0),
        AVIIF_KEYFRAME, 0, 0);
}
```

Before the program ends, we have to close the streams and the file:



```
public void closeAll()
{
    int result = AVIStreamRelease(
        psComp);
    result = AVIStreamRelease(pStream);
    result = AVIFileRelease(pFile);
}
```

The API functions are defined as:

```
[DllImport("avifl32.dll")]
extern static int
    AVIStreamRelease(IntPtr ppavi);
[DllImport("avifl32.dll")]
extern static int
    AVIFileRelease(IntPtr pfile);
```

With the class fully defined, we can use it to create a video. Add a button to the form and in its click event handler write:

```
BmpToAVI movie = new BmpToAVI(
    this.Handle);
movie.FirstFrame(@"c:\test.avi",
    @"c:\1.bmp");
for (int i = 2; i <= 10; i++)
{
    movie.NextFrame(@"c:\\" + i.
        ToString() + ".bmp");
}
movie.closeAll();
```

Before running the program for the first time, create 10 bitmap images called 1.bmp, 2.bmp and so on, and place them in the C:\ folder. They should be 24-bit uncompressed .BMP format files; you can use Paint to create them. Compressed bitmap files don't work with Vfw unless you uncompress them first, and 256-colour bitmaps don't work, but they don't generate an error message either. Bear in mind that some graphics programs don't write all the fields in the bitmap file header, and this can cause problems.

When you press the button, the 10 bitmaps are converted to a single second of video. Other parts of the Vfw API can be used in a similar way to do such things as extract frames, recompress video data, add soundtracks and so on. **CS**

CONTACT

MIKE JAMES

Mike has written several books on computing and programming and has been a senior lecturer at Teesside University

mike@computershopper.co.uk